Using Cloudscape with Database Design Patterns

James W. Cooper

Java is portable, right? Write once, run mostly anywhere, right? But one persistent problem has been one of the portability of the underlying system tools you need to build complete applications on either the client or the server. And one of the biggest headaches is handling databases, which despite their ubiquity have a number of annoying platform or price restrictions. At the low end is Microsoft Access, which while perfectly capable for small applications, is plagued by being Windows only and emphatically not multi-user. In the mid range, you have Microsoft SQL Server, which is certainly multi-user, but is still Windows only and not inexpensive. At the high end, industrial strength systems like DB2 and Oracle cover most platforms, but are kind of expensive and overkill for simpler applications.

Here is where the remarkable Cloudscape package comes into play. Cloudscape is an all-Java database system available from Informix that runs anywhere you can run a JVM. Not only does this mean any Unix as well as any Windows system, it means the Mac as well. One important market this make available is the sophisticated user application that needs a database to keep track of information. This could be customer order info or a complete mailing list system, or a system that keeps track of domain specific information like athletic times or genealogy information. And any system you built would run anywhere people want it to. It's easy to provide it to other potential users. You would not be limited to PCs or Macs or Linux. Finally you can write a truly cross-platform user application!

Of course these generalizations also apply to server systems. You can do JSP-based lookups of any kind of information on your server, and be sure that the system will port to other platforms without any significant reprogramming. This can be ideal for locally developed systems that other related groups want to implement without the interference of religious issues. Finally, if you have written your server applications in Java, you have a clear migration path to one of the industrial strength database systems when your system demands increase.

Trying Out Cloudscape

Cloudscape is available for free download in a complete, working version. It is free for personal use and for development. When you are ready to deploy a Cloudscape-based system, they will provide a package price that depends on the number of users and systems you will be deploying.

I got a copy of the free Cloudscape 3.6 CD-ROM and installed it on my Windows-2000 laptop in just a few minutes. The only confusion that arose was a requirement that the directory path not contain any spaces. You can work around this manually, however. Once Cloudscape is installed, you find that it comes with excellent documentation and a very well-written and designed tutorial, as well as example code you can run on the spot and see how Cloudscape works. I read the whole tutorial, but it didn't take long to realize

that I was going to be able to write code to build and use Cloudscape very quickly indeed because of the Design Patterns we've discussed earlier.

You can access Cloudscape directly through its SQL interface and that is what I did in writing the code for this article. In fact, I really spent a lot more time making a user interface to display my examples than I did programming to Cloudscape, because everything worked right out of the box, the first time. In addition to the SQL interface, Cloudscape comes with Cloudview, which allows you to view and modify any database from a convenient set of GUI windows.

The Façade Design Pattern

I originally discussed the Façade pattern in December of 1998, and in more detail in my book on Java Design Patterns. Then I extended its use in an article in October of 2000. Don't worry if you can't lay your hands on these issues right away, all of the example code is available for download with this article.

The Façade Design Pattern simply wraps some fairly tricky-to-use classes in some simpler ones, perhaps with a little bit less flexibility. The idea is really just another instance of the old 80/20 rule. Most users use only about 20% of the function, so why not make it easier for them. The more elaborate features are used quite seldom, but are available by going to the lower level interface when needed.

In the case of the java.sql.* package, the code for accessing databases and getting results is just a little too hard to use. It requires understanding about seven different classes, when most of the time two or three would seem to all you need. So, I wrapped this complexity into three main classes:

- Database – handles connection to the database itself
- SqlQuery – packages each query and returns a Results class.
- Results – returns the rows from a query

Then, in the October, 2000 column, I expanded on this to the case where we wanted to create new tables in addition to querying existing data, by creating a DBTable class that allows you to create tables and add columns and indexes to them, and then add data to the tables. In essence, it generates the SQL CREATE TABLE statement. The important method is shown below:

```
public void createTable() throws SQLException {
        if (exists() )  //delete previous table
            delete();
        //generate SQL string to make the table
        String sql = "CREATE TABLE " + tableName +" (" ;
        sql += getFieldString();    //get column names
        if (primaryKey.length () > 0) {
            sql += ", PRIMARY KEY(" + primaryKey +")";
        }
        sql +=")";
        SqlQuery qry = new SqlQuery(sql, db);

        qry.executeUpdate ();
        tableCreated = true;
```

```
    }
```

The getFieldString method generates a list of the columns and types to be generated from the columns you defined.

Suppose we want to create a database containing 3 tables, Stores, Foods and FoodPrices. The method for creating the Stores table is shown below using the DBTable class:

```
private void makeStores() throws SQLException {
        System.out.println("creating: Stores");
        stores = new DbTable("Stores", db);
        stores.addField ("Storekey");
        stores.addField("StoreName", 20);
        stores.setPrimaryKey ("Storekey");
        stores.createTable ();
        stores.openTable ();

        int i = 1;
        addStore(i++, "Village Market");
        addStore(i++, "Stop and Shop");
        addStore(i++, "Waldbaums");
    }
```

and the addStore method creates each row:

```
private void addStore(int key, String name) throws SQLException {
        System.out.println("adding:" + name);
        stores.setPreparedTerm (key, 1);
        stores.setPreparedTerm (name, 2);
        stores.addRow();
    }
```

Now, back to Cloudscape. The good news is that all of the code from these previous two articles worked without any significant change. The only thing I found is that Cloudscape's SQL parser is not happy with semicolons at the end of SQL statements. Since they are optional in other systems, I just deleted them from the DBTable class.

In the same way, I added some grocery names to a second table:

```
private void makeGrocs() throws SQLException {
        System.out.println("creating: Food");
        grocs = new DbTable("Food", db);

        grocs.addField ("Foodkey");
        grocs.addField("Foodname", 20);
        grocs.setPrimaryKey ("Foodkey");
        grocs.createTable ();
        grocs.openTable ();

        int i = 1;
        addGroc(i++, "Apples");
        addGroc(i++, "Oranges");
        addGroc(i++, "Milk");
        addGroc(i++, "Butter");
        addGroc(i++, "Green beans");
        addGroc(i++, "Cola");
        addGroc(i++, "Hamburger");
```

```
    }
```

and made a price table in the same way using the StoreKey, the FoodKey and a price value as shown in Table 1.

FSKey	StoreKey	FoodKey	Price
1	1	1	$0.27
2	2	1	$0.29
3	3	1	$0.33
4	1	2	$0.36
5	2	2	$0.29
6	3	2	$0.47
7	1	3	$1.98
8	2	3	$2.45
9	3	3	$2.29

Table 1 – A portion of the FoodPrice table.

Running the Program

Our program creates the database each time and allows you to query the list of groceries and select one to get the prices. Using the methods we have already shown, the database creation step becomes:

```
//creates the tables in the database
    private void createDB() throws SQLException {
       Cursor wait = new Cursor(Cursor.WAIT_CURSOR);
        setCursor(wait);
        db.create("GroceryDB"); //Create the databse
        makeGrocs();        //groceries table
        makeStores();       //stores table
        makePrices();       //food price table
        Cursor def = new Cursor(Cursor.DEFAULT_CURSOR);
        setCursor(def);
        db.close ();        //close
    }
```

You see the program after the grocery table is created in Figure 1.

The query to load the table when you click on the Groceries button is:

```
private void showGrocs() {
        try {
            db.Open ("GroceryDB");
            String sql="Select FoodName from Food order by FoodName";
            SqlQuery qry = new SqlQuery(sql, db);
            Results rs = qry.Execute ();
            while (rs.hasMoreElements ()) {
                jlist.add (rs.getColumnValue (1));
            }
        } catch (SQLException e) {
            System.out.println(e.getMessage ());
            e.printStackTrace ();
```

```
        }
    }
```

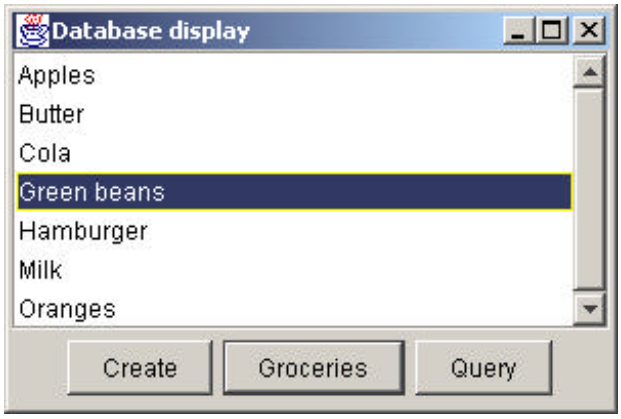

Figure 1 – The grocery list.

When we click on the Query button, we show the query window and issue the query

```
String queryText ="SELECT FoodName, StoreName, Price "+
 "FROM (Food INNER JOIN FoodPrice ON " +
 "Food.FoodKey = FoodPrice.FoodKey) " +
 "INNER JOIN Stores ON " +
 "FoodPrice.StoreKey = Stores.StoreKey "+
 "WHERE (((Food.FoodName)=\'" + produce+"\')) " +
 "ORDER BY FoodPrice.Price";
 try {
   SqlQuery qry = new SqlQuery(queryText, db);
   Results rs = qry.Execute ();
   while (rs.hasMoreElements () ) {
     Double price = new Double(rs.getColumnValue (3));
     DecimalFormat form = new DecimalFormat();
     form.applyPattern ("##.00");
    jlist.add (rs.getColumnValue (2)+" "+
          form.format(price.doubleValue ()));
   }
  }
  catch(SQLException e) {
    System.out.println(e.getMessage());
 }
}
```

The results are shown in Figure 2.

Figure 2 – The results of the price query.

To see what we've created, I ran Cloudview to display the database tables (Figure 3) and the contents of one of the tables (Figure 4).

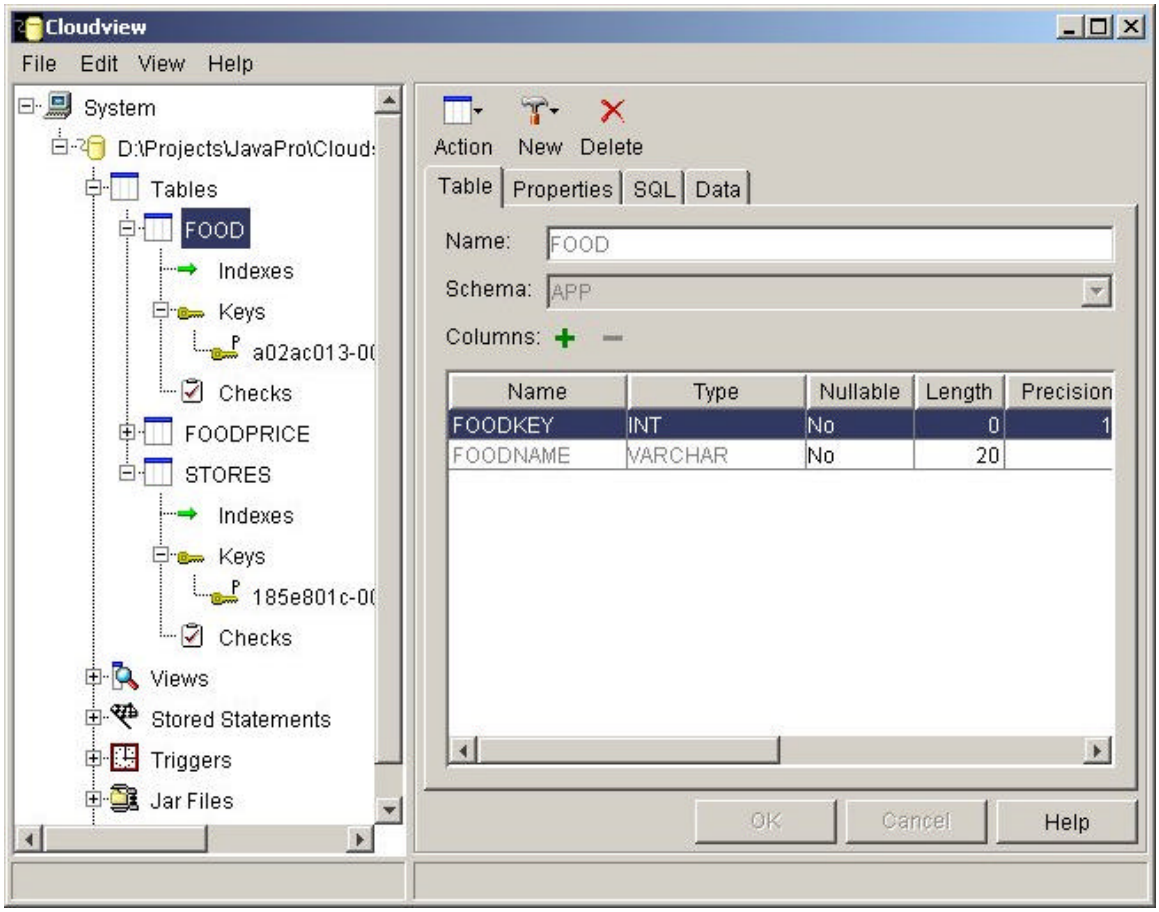

Figure 3 – The Food database table structure, shown in Cloudview

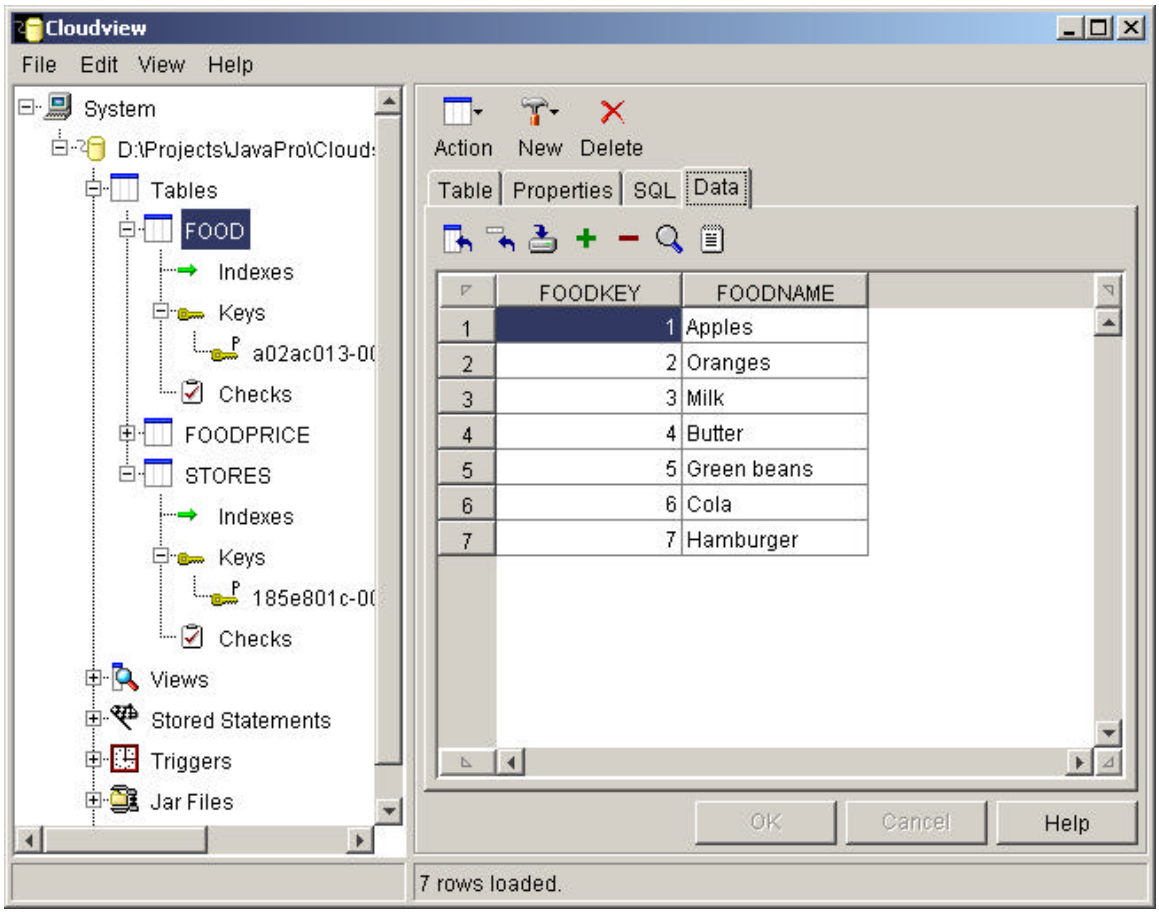

Figure 4 – The contents of the Food database table.

What Did I have to Change?

I really only changed two things to get Cloudscape working with my existing Façade code. First, I removed the semicolons from the end of all the SQL the classes generate. And second, I derived the CloudDatabase class from my generic Database class, to allow for the ability to create a database using a special connect string that Cloudscape supports:

```
//Connects to Cloudscape databases
public class CloudDatabase extends Database {
    public CloudDatabase()
    throws ClassNotFoundException {
        super("COM.cloudscape.core.JDBCDriver");
    }
    //----------------------------
    public void create(String databaseName)
    throws SQLException {
        super.Open ("jdbc:cloudscape:"+
                    databaseName+
                    ";create=true", null);
    }
    //----------------------------
    public void Open(String databaseName)
    throws SQLException {
```

```
        super.Open ("jdbc:cloudscape:"+
                    databaseName+
                    ";", null);
    }
    //----------------------------
    public void close() {
        boolean gotSQLExc = false;
        try {
            DriverManager.getConnection(
                "jdbc:cloudscape:;shutdown=true");
        } catch (SQLException e) {
            gotSQLExc = true;
        }
        catch(Exception e) {
            System.out.println("Abnoraml shutdown:"
                               +e.getMessage());
        }
    }
}
```

Other than that, Cloudscape and my classes worked together right from the beginning. I built a scalable cross-platform database solution in just a few minutes. I think this is pretty impressive. Cloudscape will handle multiple connections and they provide some example JSPs for using it in a client-server environment. And since it is a native Java system, you can include stored Java object instances and stored Java procedures. I liked that a lot. And finally, should I need to scale to some humongous multi-server solution with terabytes of data, I can just plug in DB2 instead of Cloudscape without changing anything except the version of the Database class I use. I'm very impressed.